NUEVOS APIS DE PROGRAMACIÓN

Celeste Campo
celeste@it.uc3m.es

Parte de este material se basa en transparencias de Natividad Martínez Madrid (nati@it.uc3m.es)

ÍNDICE

- Mobile Media API (MMAPI).
- Nuevas APIs y extensiones de MIDP.

BIBLIOGRAFÍA

- *J2ME : Java 2 micro edition : manual de usuario y tutorial*. Froufe, Agustín y Jorge, Patricia. Ra-Ma. [2004]. L/S 004.438 JAVA FRO, L/D 004.438 JAVA FRO.
- http://developers.sun.com/techtopics/mobility/apis/articles/mmapioverview/

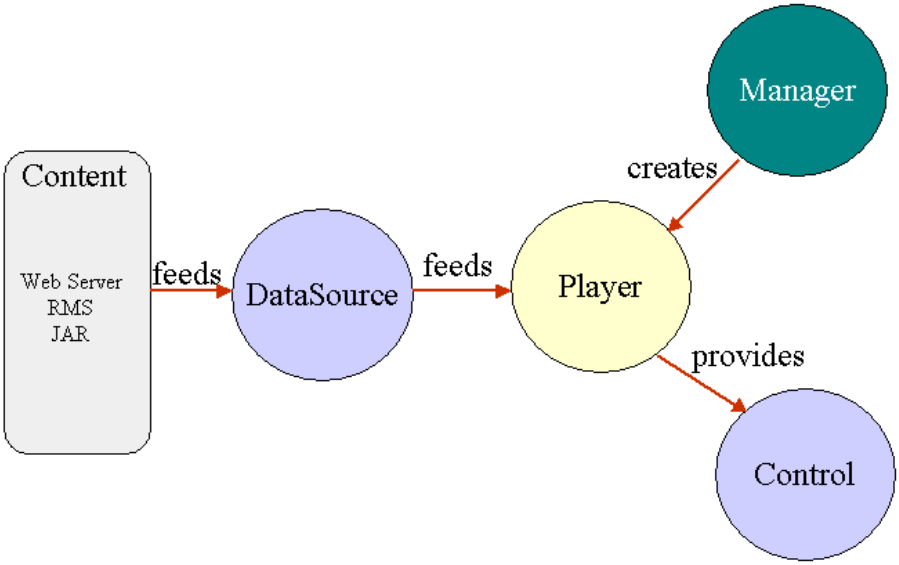
MMAPI - Mobile Media API

- Paquete opcional que proporciona soporte para la generación, reproducción y grabado de datos multimedia (audio o video).
- Definido en el JSR 135:
  - Tres nuevos paquetes:
    - **`javax.microedition.media`**
    - **`javax.microedition.media.control`**
    - **`javax.microedition.media.protocol`**
  - MIDP 2.0 soporta un subconjunto de este API:
    - Reproducción de audio.
- Principales características:
  - Footprint reducido: puede ejecutarse en dispositivos con restricciones de memoria (dispositivos CLDC).
  - Independiente del tipo de contenido o del protocolo utilizado.
  - Extensible.
  - Puede soportar sólo un subconjunto (por ejemplo, audio básico).
  - Facilita a los que implementan el API incluir sólo determinada funcionalidad.

MMAPI - Procesado multimedia

- Partes del procesamiento multimedia:
  - *Gestión del protocolo*: lectura de datos de una fuente, como un fichero o un servidor de streaming, al sistema de procesado multimedia.
  - *Gestión del contenido*: parseado o decodificación de los datos multimedia y reproducción al dispositivo de salida, como altavoz audio o pantalla video.
- Para facilitar estas operaciones el API proporciona:
  - **`DataSource`**
    - Encapsula y abstrae la gestión del protocolo, ocultando los detalles de cómo se leen los datos de la fuente.
    - Permite al objeto **`Player`** gestionar el contenido.
  - **`Player`**
    - Lee los datos de **`DataSource`**.
    - Los procesa.
    - Los manda al dispositivo de salida.
    - Incluye métodos para el control de la reproducción.
  - **`Manager`**:
    - Permite crear **`Player`**s a partir de **`DataSource`**s

MMAPI - Arquitectura



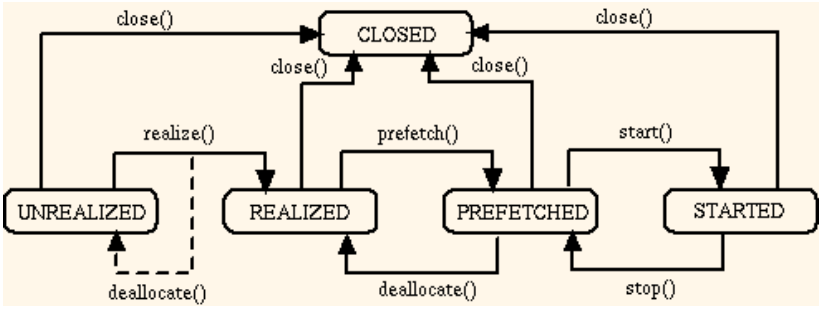
MMAPI – Clase `Manager`

- Punto de partida para procesar o reproducir datos multimedia:
  - Reproducción de tonos simples.
  - Creación de objetos Player.
- Reproducción de tonos simples:
  - **`playTone(int nota, int duration, int volumen)`**
- Clase factoría de Players
  - **`Player player = Manager.createPlayer(String url);`**
  - URL: **`<protocol>:<content location>`**
  - La aplicación usa los métodos del Player creado para controlar la descarga y reproducción del datos multimedia

MMAPI - Estados de `Player`

- El ciclo de vida de un **`Player`** incluye cinco estados, y seis de los métodos de la clase **`Player`** controlan las transiciones entre estos estados:
  - **`UNREALIZED`**
    - Tras su creación.
  - **`REALIZED`**
    - Tras llamada a **`realize()`**, inicializa la información para adquirir recursos multimedia.
  - **`PREFETCHED`**:
    - Tras llamar a **`prefetch()`**, establece la conexión de red para el streaming de datos y otras inicializaciones.
  - **`STARTED`**:
    - Tras la llamada a **`start()`**, el player puede procesar los datos
    - Al acabar el procesado (fin del stream multimedia) vuelve al estado **`PREFETCHED`**.
  - **`CLOSE`**
    - Tras la llamada a **`close().`**

MMAPI - Estados de `Player`



MMAPI - Interfaz `Control`

- Un **`Player`** proporciona control específico para el tipo de medio que procesa:
  - **`getControl()`** para un único control.
  - **`getControls()`** para un array de controles.
  - Ejemplo:
    - Un player de MIDI que invoque **`getControl()`** recibe un **`MIDIControl`**.

MMAPI - Tipos de controles

- Tipos de controles definidos en **`javax.microedition.media.control`**:
  - **`FramePositionControl`**: posición de un video frame para un Player.
  - **`GUIControl`**: soporte de componentes GUI.
  - **`MetaDataControl`:** para obtener información de metadatos en un stream multimedia.
  - **`MIDIControl`**: acceso a dispositivos de reproducción y transmisión de MIDI.
  - **`PitchControl`**: controla pitch de reproducción de salida audio sin cambiar velocidad de reproducción.
  - **`RateControl`**: controla la tasa de reproducción.
  - **`RecordControl`**: controla la grabación multimedia.
  - **`StopTimeControl`**: para predeterminar tiempo de parada de un player.
  - **`TempoControl`**: controla el tempo de una canción, en players para ficheros MIDI.
  - **`ToneControl`**: reproducción de secuencia de tonos predefinidos por el usuario.
  - **`VideoControl`**: controla la reproducción de video (localización relativa al canvas donde se visualiza, etc.).
  - **`VolumeControl`**: volumen de un Player.

MMAPI - Consideraciones

- MMAPI no obliga a implementar un determinado tipo de medio ni de protocolo, cada implementación debe decidir cuál de ellos se soporta:
- Se debe garantizar soporte al menos a un tipo de medio y un protocolo.
- Para obtener los tipos y protocolos que soporta una implementación determinada, se proporcionan los métodos:
  - **`Manager.getSupportedContentTypes()`**
  - **`Manager.getSupportedProtocols()`**

MMAPI - Ejemplos

- Reproducir un tono:
  - Un tono está caracterizado por su frecuencia y su duración.

```
...
try {
  // reproduce un tono durante 4000 mseg. a un volumen 100
 Manager.playTone()(ToneControl.C4, 4000, 100);
}
  catch(MediaException me) {
}
...
```

MMAPI - Ejemplos

- Reproducción MP3

```
...
try {
  Player p =   Manager.createPlayer("http://server/
                                        somemusic.mp3");
  p.setLoopCount(5);
  p.start();
}
  catch(IOException ioe) {
}
  catch(MediaException e) {
}
...
```


MMAPI - Ejemplos

- Reproducción de MP3 con controles

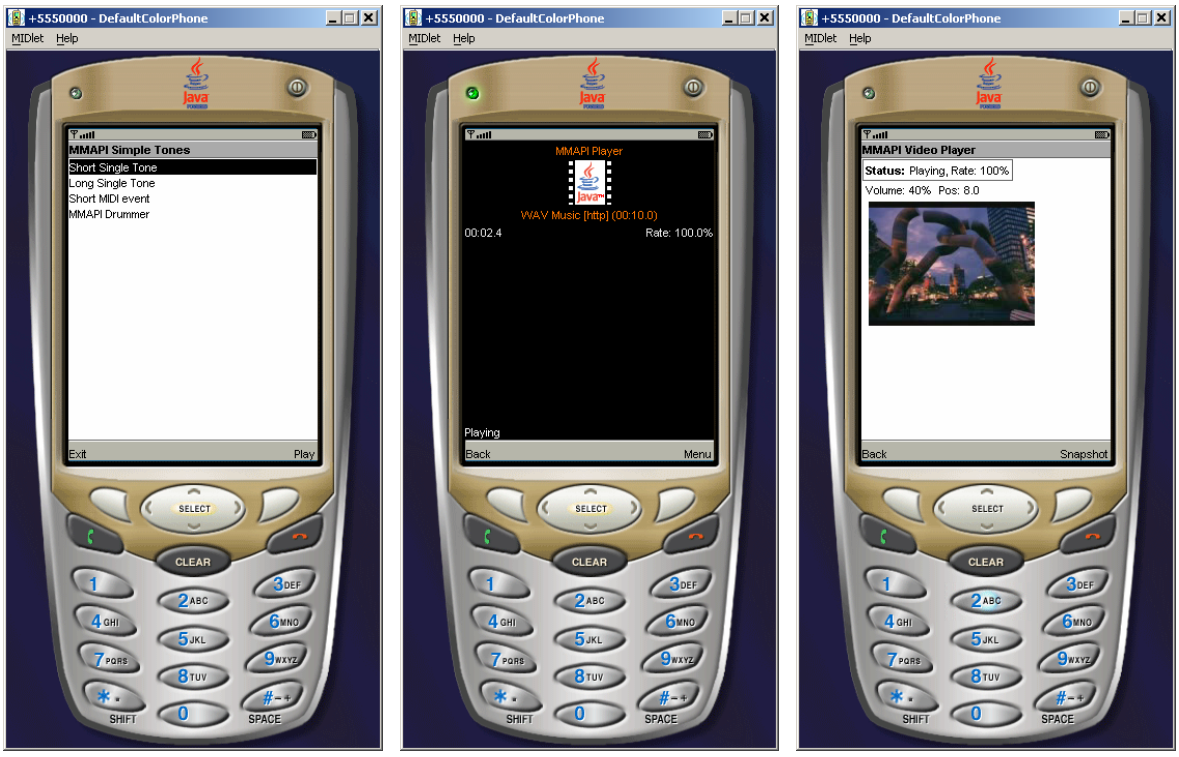
```
...
Player p;
VolumeControl vc;
try {
  p = Manager.createPlayer("http://server/somemusic.mp3");
  p.realize();
  // Obtiene un VolumeControl para el reproductor
  // y pone el volumen al máximo
  vc = (VolumeControl)p.getControl("VolumeControl");
  if (vc != null) {
    vc.setVolume(100);
  }
  // El reproductor puede empezar con una pequeña latencia
  p.prefetch();
  // Arranque no bloqueante
  p.start();
}
  catch(IOException ioe) {
}
  catch(MediaException e) {
}
...
```

MMAPI - Ejemplos

- Reproducción de vídeo:

```
...
Player p;
VideoControl vc;
try {
  p = Manager.createPlayer("http://server/somemovie.mpg");
  p.realize();
  // obtiene VideoControl
  vc = (VideoControl)p.getControl("VideoControl");
  ...
  p.start();
}
  catch(IOException ioe) {
}
  catch(MediaException me) {
}
...
```


MMAPI – WTK mmademo



Nuevas APIs y extensiones de MIDP

- APIs y extensiones existentes:
  - Java Technology for the Wireless Industry (JSR 185):
    - Específico para teléfonos móviles de siguiente generación.
  - Wireless Messaging API (JSR 120, JSR 205):
    - Acceso a sistemas de envío de mensajes (SMS).
  - Security and Trust Services (JSR 177):
    - Mejora la seguridad añadiendo APIs criptográficas, firmas digitales, gestión de credenciales.
  - Web Services APIs (JSR 172):
    - Desarrollo de clientes Web en dispositivos móviles (web services).
  - Bluetooth API (JSR 82):
    - Desarrollo de aplicaciones que usan Bluetooth.
  - Location API (JSR 179):
    - Acceso a información de localización
  - SIP API (JSR 180):
    - Desarrollo de clientes SIP.
  - Mobile 3D Graphics (JSR 184):
    - Desarrollo de juegos 3D.

Nuevas APIs y extensiones de MIDP

- Todas las extensiones de MIDP se definen a través de Java Community Process (http://jcp.org).
- ¿Cómo puedo saber si un dispositivo soporta una extensión?
  - Necesario ir a las especificaciones técnicas de los fabricantes del dispositivo.
  - Realizar pequeños programas de prueba para saber cuales o qué parte de las extensiones soporta.
- ¿Cómo puedo encontrar ejemplos y ayuda para desarrollar un MIDlet utilizando estas extensiones?:
  - http://developers.sun.com/techtopics/mobility/reference/techart/index.html

CONSEJOS DE PROGRAMACIÓN

Celeste Campo
celeste@it.uc3m.es

ÍNDICE

- Uso de memoria.
- Uso de threads.

BIBLIOGRAFÍA

- http://developers.sun.com/techtopics/mobility/midp/ttips/appsize/
- http://developers.sun.com/techtopics/mobility/midp/articles/threading2/

Introducción

- Al desarrollar software para dispositivos limitados es muy importante realizar optimizaciones siempre que sea posible.
- Algunos de consejos de programación que vamos a ver se centran en:
  - Uso de memoria.
  - Uso de threads.
- Otros consejos de programación los podéis obtener en:
  - http://developers.sun.com/techtopics/mobility/allttips/#midp

Uso de memoria

- Los sistemas J2ME tienen restricciones importantes de memoria para el almacenamiento y ejecución de aplicaciones:
  - Restricciones por debajo de 50K.
  - Aunque el dispositivo tenga mucha más memoria, se limita la que se puede utilizar para aplicaciones J2ME.

Uso de memoria

1. Eliminar todas las clases que no son necesarias:
   - ¿Hemos incluido funcionalidad que no es necesaria?
2. Analizar todas las clases internas que utiliza la aplicación y reducir su número:
   - Cada clase que se define introduce una sobrecarga.
3. Maximizar el uso de clases existentes y no crear clases propias.
4. Disminuir los niveles de jerarquía de clases:
   - Compromiso entre modularidad de código y tamaño.
5. Utilizar nombre cortos para los paquetes, clases, métodos y miembros de clase.
   - Intentando no perder la legibilidad del código.
   - Herramientas que ofuscan el código, realizan esta tarea automáticamente.
6. Inicializaciones de arrays:
   - Introducir datos en un string y convertirlo a un array en tiempo de ejecución.
   - Obtener los datos desde un fichero binario empaquetado con la aplicación.

Uso de Threads

- Permiten a la aplicación realizar múltiples actividades simultáneamente:
  - El interfaz de usuario se mantiene activo mientras se realizan operaciones largas (comunicación por red o cálculos complicados)
- ¿Qué es un thread?
  - Un flujo de control secuencial individual dentro de un programa.
  - Cada aplicación en ejecución tiene al menos un thread. Si tiene más, se llama multithreaded.

Uso de Threads – Contexto

- El contexto del thread mantiene información como:
  - Dirección de la instrucción actual.
  - Almacenamiento para variables locales.
  - Estado del thread.
- El estado del thread puede ser:
  - *Running*: en ejecución.
  - *Ready*: listo para ejecutar.
  - *Suspended*: esperando a un evento externo.
  - *Terminated*: finalizada la ejecución.
- Thread scheduling (planificación de hilos):
  - Cambio de contexto (context switching) puede suceder en cualquier momento.
  - Controlado por el sistema, el programador sólo puede indicar un nivel de prioridad (con mayor prioridad se ejecutará más frecuentemente que con menor prioridad).

Uso de Threads – Clase `Thread`

- **`java.lang.Thread`**
- Dos versiones en J2ME:
  - CDC: compatible con J2SE
  - CLDC: versión reducida
    - **`activeCount()`**
    - **`currentThread()`**
    - **`getPriority()`**
    - **`isAlive()`**
    - **`join()`**
    - **`run()`**
    - **`setPriority()`**
    - **`sleep()`**
    - **`start()`**
    - **`yield()`**

Uso de Threads – Arranque

- Para arrancar un nuevo thread hace falta:
  - Un objeto que implemente el interfaz java.lang.Runnable (define run())

```
public class DoSomething implements Runnable
  public void run(){
  // do something
  }
}
```

- Con ese objeto, se crea un objeto Thread y se invoca su método **`start()`**:

```
DoSomething doIt = new DoSomething();
Thread myThread = new Thread(doIt);
myThread.start();
```

- Cuando el thread vuelve del **`run()`**, el sistema termina el thread.
- Podría crearse una clase que extienda de Thread, pero siguiendo los consejos de uso de memoria en J2ME, esto (implementar **`Runnable`**) es más eficiente.

Uso de Threads – Parada

- No existe el método **`stop()`** y el método **`interrupt()`** sólo existe en CLDC 1.1.
- No se puede parar un thread desde fuera, se le puede “indicar” que pare:

```
public class MyThread implements Runnable
  private boolean quit = false;
  public void run(){
    while (!quit){
      //do something
    }
  }
  public void quit(){
    quit = true;
  }
}
```

- Si el thread usa recursos del sistema (conexiones de red o Record Store) las debe liberar antes de terminar (usar **`finally`**):
  - Importante también porque en algunos dispositivos se limita el número de accesos simultáneos a un tipo de recurso.
- Uso de **`isAlive()`** para comprobar si aún se está ejecutando, y de **`join()`** para activarse cuando el otro termine.

Uso de Threads – Sincronización

- Como los cambios de contexto pueden ocurrir en cualquier instante, debe garantizarse sincronización para controlar el acceso a datos compartidos:
  - Uso de monitores.
  - Cualquier objeto Java puede ser un monitor:
    - El mismo objeto (**`this`**).
    - Utilización de la palabra reservada **`synchronized`**.

```
public int increment() {
  synchronized( this ) {
    return ++counter;
  }
}
```

```
public synchronized int increment() {
  return ++counter;
  }
```


Uso de Threads – Sincronización

- Sincronización de Threads se puede realizar a través de clases que utilizan métodos **`synchronized`**.
- Recomendaciones:
  - Utilizar clases inmutables:
    - Su estado no cambia después de la inicialización (**`String`**).
  - Evitar el *deadlock*:
    - El thread A bloquea el objeto X e intenta acceder al Y. El thread B bloquea el objeto Y e intenta acceder al X
    - Bloquear los objetos siempre en el mismo orden: siempre bloquear X antes de Y.

Uso de Threads – Espera y notificaciones

- En muchas ocasiones, los threads esperan algún evento antes de acceder a los datos:
  - Ej: lectura de una cola de mensajes escritos por otros threads.
- Espera activa (*busy wait*):
  - El thread está todo el rato comprobando la aparición del evento.
  - Consume muchos recursos.
- Espera suspendida (*suspended wait*):
  - El thread se suspende hasta que ocurra el evento.
  - Se recomienda su uso en J2ME porque consume menos recursos.

Uso de Threads – Espera suspendida

- El mismo monitor que se usa para la sincronización se puede usar para la suspensión:
  - Métodos **`wait()`**, **`notify()`** y **`notifyAll()`** de **`Object`**
- El thread se suspende a sí mismo llamando al método **`wait`** del objeto monitor

```
synchronized(obj){
// comprueba el estado del objeto
  try{
    obj.wait();
  } catch( InterruptedException e){
  }
}
```

- Opcionalmente se puede especificar un timeout en **`wait`** (en mseg.)

Uso de Threads – Espera suspendida

```
import java.util.Vector;

public class Worker implements Runnable{
  private boolean quit = false;
  private Vector queue = new Vector();

  public Worker(){
    new Thread (this).start();
  }
```


Uso de Threads – Espera suspendida

```
  public void run(){
    Object o;

    while (!quit){
      o=null;
      synchronized (queue) {
        if (queue.size() > 0){
          o=queue.elementAt(0);
          queue.removeElementAt(0);
        } else {
          try{
            queue.wait();
          } catch (InterruptedException e){ }
        }
      }
    if (o!= null) {
      // do something
    }
   }
  }
```

Uso de Threads – Espera suspendida

```
  public boolean addToQueue( Object o ){
    synchronized( queue ){
        if( !quit ){
          queue.addElement( o );
          queue.notify();
          return true;
        }
        return false;
    }
  }

    public void quit(){
      synchronized( queue ){
        quit = true;
        queue.notify();
      }
    }
}
```


Uso de Threads – Consideraciones

- Como regla general no se debe suspender un thread que no ha sido creado por tu aplicación:
  - Por ejemplo, los threads del sistema encargados de la gestión del interfaz gráfico y de eventos.
  - Porque provocaría que la aplicación se bloquease e interfiriese en otras aplicaciones.